Research Paper ■

Fast Exact String Pattern-matching Algorithms Adapted to the Characteristics of the Medical Language

CHRISTIAN LOVIS, MD, ROBERT H. BAUD, PHD

Abstract **Objective:** The authors consider the problem of exact string pattern matching using algorithms that do not require any preprocessing. To choose the most appropriate algorithm, distinctive features of the medical language must be taken into account. The characteristics of medical language are emphasized in this regard, the best algorithm of those reviewed is proposed, and detailed evaluations of time complexity for processing medical texts are provided.

Design: The authors first illustrate and discuss the techniques of various string pattern-matching algorithms. Next, the source code and the behavior of representative exact string pattern-matching algorithms are presented in a comprehensive manner to promote their implementation. Detailed explanations of the use of various techniques to improve performance are given.

Measurements: Real-time measures of time complexity with English medical texts are presented. They lead to results distinct from those found in the computer science literature, which are typically computed with normally distributed texts.

Results: The Boyer-Moore-Horspool algorithm achieves the best overall results when used with medical texts. This algorithm usually performs at least twice as fast as the other algorithms tested.

Conclusion: The time performance of exact string pattern matching can be greatly improved if an efficient algorithm is used. Considering the growing amount of text handled in the electronic patient record, it is worth implementing this efficient algorithm.

■ **J Am Med Inform Assoc.** 2000;7:378–391.

Many, if not most, sources of medical texts are preprocessed and allow fast queries. However, health care providers and scientists are doing more and more literature research and text queries using the Internet. Information that comes through this way is not preprocessed and usually arrives as full-streamed text, in real time, in the user's computer. Clinicians need help filtering the information they receive from Internet sources like MEDLINE. On the one hand, too-specific queries will induce silence that is difficult to detect. On the other hand, too-sensitive queries provide huge numbers of data that cannot be reviewed in clinical settings in a reasonable time. A local filter providing specificity to a sensitive query in real time can answer this problem.

The same needs of filtering streamed data occur when information about patients is received from several electronic patient records (EPRs) and has to be organized in real time. Natural language processing (NLP) technologies are in focus in the literature, especially because of the growing importance of textual medical information.[1–3] The shift in the trend from numeric

Affiliations of the authors: Puget Sound Health Care System, Seattle, Washington (CL); University Hospital of Geneva, Geneva, Switzerland (RHB).

This work was supported by a grant from the University Hospital of Geneva, Switzerland.

Correspondence and reprints: Christian Lovis, MD, University Hospital of Geneva, Division of Medical Informatics, Rue Micheli-du-Crest, CH-1211 Geneva 4, Switzerland;
e-mail: ⟨christian.lovis@dim.hcuge.ch⟩.

Received for publication: 10/26/99; accepted for publication: 2/16/00.

data to textual data is one of the paradigms of the modern EPR. Command-line order-entry systems are gaining popularity, and "just-in-time" literature retrieval in the EPR is becoming an essential tool.[4]

The amount of available textual data tends to double at an increasing rate. Powerful commercial document management systems that will enhance this trend are now available. With the expanding computational power at the workplace, the use of free-text data entry systems allows more freedom for physicians while maintaining the capability of further processing. In addition, the EPR extends increasingly to other sources of information, such as online medical textbooks and Internet-based references.[5]

Recent developments made in the EPR at the University Hospital of Geneva have shown the necessity of parsing, analyzing, and organizing medical text of various sources in real time during the clinician's work. A review of the medical informatics literature on that subject shows that little attention has been given to string pattern matching, especially because research in NLP has been highly focused on semantic representation, which is considered the most important and difficult step. Hume and Sunday[6] state that, "partially because the best algorithms presented in the literature are difficult to understand and to implement, knowledge of fast and practical algorithms is not commonplace." Other authors report similar observations.[7,24]

Two very different groups of techniques are known in the domain of string pattern matching. One deals with exact string pattern matching, while the other deals with partial or incomplete string pattern matching. The latter group addresses automatic word correction and the identification of typographic errors. Assuming that the pattern length is no longer than the unit memory size of the machine, the shift-or algorithm is an efficient algorithm that adapts easily to a wide range of approximate string matching problems. This algorithm is based on finite automata theory, such as the Knuth-Morris-Pratt algorithm, and exploits the finiteness of the alphabet, as in the Boyer-Moore algorithm.

In pattern recognition works, genetic algorithms are widely used to identify occurrences of complex patterns, although they are regarded primarily as a problem-solving method. Genetic algorithms are based on ideas from population genetics; they feature populations of genotypes (characteristics of an individual) stored in memory, differential reproduction of these genotypes, and variations that are created in a manner analogous to the biological processes of mutation and crossover. Genetic algorithms are powerful tools for solving complex pattern-matching problems, especially when the matching is incomplete or inexact or when it occurs on repetitive patterns separated by unmatched patterns, as it can be in searches for long DNA sequences that take into account possible alterations, from single deletions or insertions to crossovers.[8,9] However, these issues will not be discussed here.

Most exact string pattern-matching algorithms are easily adapted to deal with multiple string pattern searches or with wildcards. We present the full code and concepts underlying two major different classes of exact string search pattern algorithms, those working with hash tables and those based on heuristic skip tables. Exact string matching consists of finding one or, more generally, all of the occurrences of a pattern in a target. The algorithmic complexity of the problem is analyzed by means of standard measures of the running time and amount of memory space required by the computations.

Text-based applications must solve two kinds of problems, depending on which string, the pattern or the target, is given first. Algorithms based on the use of automata or combinatorial properties of strings are usually implemented to preprocess the pattern and solve the first kind of problem. However, difficulties arise when one tries to recognize a specific pattern in various targets or streams of symbols arriving through a communication channel. In this case, no preprocessing of the target is possible. This contrasts with locating words in a dictionary or an entire corpus of texts that are known in advance and can be preprocessed or indexed. Among all exact text pattern-matching algorithms, those that can be used in real time on continuous streams of data are the focus of this paper. This means that neither the target nor the patterns are known in advance. Therefore, neither preprocessing nor indexing is feasible.

Notation

The term *target* is used to specify the corpus of text to search in, and the term *pattern* to identify the substring of text to search for. The pattern is denoted by $x = x_{0..m-1}$ characters and the target by $y = y_{0..n-1}$ characters. The first character in a target is therefore y_0, and the last one is y_{n-1}, where n and m are, respectively, the length of the target and the pattern. The notation x_i or y_j refers to the characters currently analyzed in both the pattern and the target. By definition, the *alphabet* is the set of all possible symbols that can be used to represent a target or a pattern. The representation $O(mn)$ is used to express the time complexity of the algorithm

Table 1 ■

Effect of Multiple Occurrences on the Number of Iterations before a Correct Fail Occurs

ICD Expression	Segments	Occurrences in ICD-10	Minimum Iterations
dilated cardio-myopathy		1	
	dilated	1	7
	cardiomyopathy	27	378
	cardio	106	636
	myopathy	51	408
	myo	297	891
	pathy	354	1,770

as a function of the sizes of the pattern and the target. Typically, the pattern is small and the target is large, m being much smaller than n.

Morphologic Characteristics of Medical Language

Medical language is characteristic in its wide and frequent use of Latin and Greek roots. This results in many words that have similar suffixes or prefixes. The consequence of frequent use of common roots is that trivial string pattern-matching algorithms, which usually perform their character matching sequentially, have a high rate of unnecessary *fail* iterations.

Our previous work in morphosemantic analysis and representation of medical text demonstrates the high use rate of compound words in medical texts.[10,11] The frequent utilization of common roots implies that, in theory, purely left-to-right or right-to-left algorithms are generally less efficient than algorithms that proceed randomly. Although these roots have a typical length of five to seven characters, their aggregation can lead to much longer repetitive patterns. Table 1 shows an example of a diagnosis found in ICD-10 with the number of occurrences of its different roots. So, for example, an algorithm analyzing strictly from right to left will have to compare the word *cardiomyopathy* 27 times before it will find the first association with *dilated*.

Some frequent patterns can be found within words that are an aggregation of typical common roots, such as *gastroentero* or *colorecto*, and suffixes, such as *ectomy* and *stomy*. The behavior of that kind of typical pattern in medical language is recognized and has been widely studied.[12–15] The overall result is a non-normal distribution of morphologic patterns in texts, which emphasizes the problem of exact string pattern-matching by increasing significantly the number of attempts needed to uniquely identify words when common algorithms are used. An example is shown in Figure 1.

Increasing the performance of exact pattern matching is a complex task. Theoretic good solutions can have high computational costs. For example, because of common prefixes and suffixes, it is interesting, in theory, to perform a pattern comparison in the middle of a potential match to improve the chance of failure in case of mismatch. However, the cost of searching the middle of a word (performing a div 2, for example) appears to be more expensive in time than performing a few additional comparisons, except in the case of long patterns.

Search Algorithms

Major categories and characteristics of exact string pattern matching algorithms have been reviewed by Hume and Sunday[16] and Pirklbauer,[17] among others. Deep and comprehensive understanding of conceptual frameworks and theoretic estimates of time and

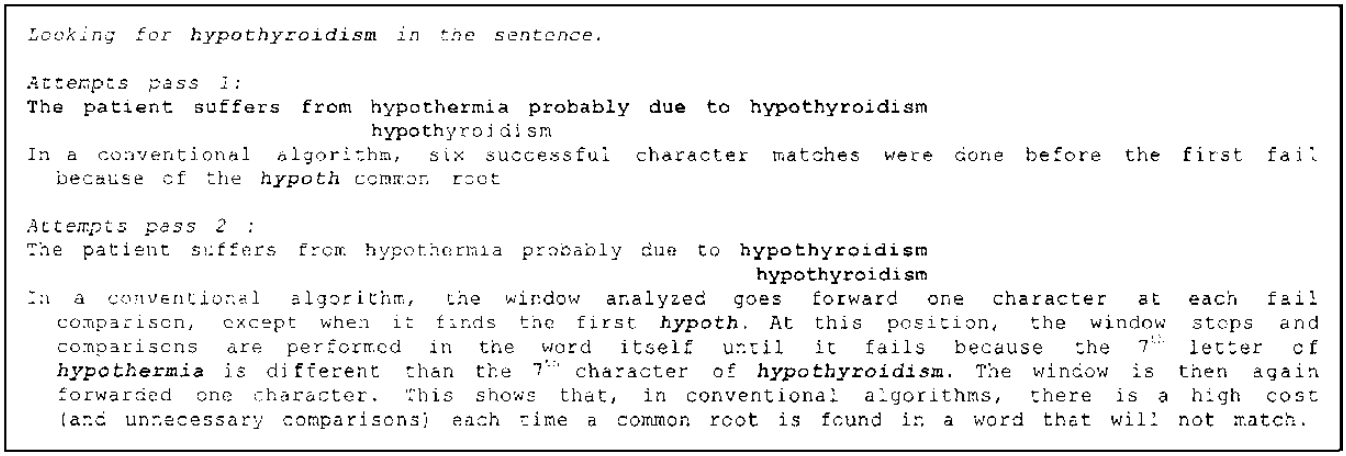


Figure 1 Example of exact pattern matching in case of common roots.

space complexity is provided,[18] especially in analyzing the behavior of these algorithms in the worst cases or using randomly generated string targets and patterns. However, there is a lack of data about the average behavior of these algorithms when used with real text corpora that have highly non-normal distributed symbols, such as the medical sublanguage. When neither preprocessing nor indexing is feasible for the target, and only limited preprocessing in space and time is affordable for the pattern, the problem of pattern matching analysis can be roughly divided into two sequences—*comparison* and *slide to next comparison*. Both steps can be optimized.

The comparison step often can be spared by the use of a hashing table.[21] Hashing functions provide an alternative way to compare strings. Essentially, this technique is based on the comparison of string hash values instead of direct character comparison and usually requires the text to be preprocessed. Hashing is a powerful and simple technique for accessing information: Each key is mathematically converted into a number, which is then used as an index into a table. In typical applications, it is common for two different strings to map to the same index, requiring some strategy for collision resolution. With a perfect function, a string could be identified in exactly one probe, without worrying about collisions.

A simple hash function performs calculations using the binary codes of the characters in a key. The record's position in the table is calculated based only on the key value itself, not how many elements are in the table. This is why hash-table inserts, deletes, and searches are rated constant time. However, finding perfect hashing functions is possible only when the set of possibilities is completely known in advance, so that each unique entry in the text has one entry in the hash table. In case of collision, that is, when the hashing value is the same for two different substrings, a formal, character-by-character comparison is performed. If the comparison is a success, a match is found. In our measures, such algorithms behave poorly for medical text, with high costs due to the time spent in the hashing function. It must be emphasized, however, that algorithms using hashing tables perform spectacularly when the target can be preprocessed and can be easily extended to multidimensional pattern matching problems. These problems are common in bioinformatics and imaging.

Use of a skip table that allows a jump of more than one character in the case of failure[20] can optimize the slide step. To understand the use of heuristic skip tables, consider that the target is examined through a window. This window delimits a region of the target and usually has the length of the pattern. Such a window slides along the target from one side to the other and is periodically shifted according to rules specific for each algorithm. When the window is at a certain position on the target, the algorithm checks whether the pattern occurs there or not by comparing symbols in the window with the corresponding aligned symbols of the pattern. If a whole match occurs, the position is reported. During this scan operation, the algorithm acquires information from the target that is used to determine the length of the next shift of the window. This operation is repeated until the end of the pattern goes beyond or reaches the end of the target. It is an effective procedure to optimize time cost during a scan operation that can be used in right-to-left or left-to-right search engines.

The so-called *naive* or *brute force* algorithm is the most intuitive approach to the string pattern-matching problem. This algorithm attempts simply to match the pattern in the target at successive positions from left to right. If failure occurs, it shifts the comparison window one character to the right until the end of the target is reached. In the worst case, this method runs in $O(mn)$ time, but in a practical approach, the expected overall performance is near $O(n + m)$. Despite the theoretic bad performance of this algorithm, our measures show that the naive algorithm is one of the fastest methods when the pattern is a short sequence of characters.

The Knuth-Morris-Pratt method[19] derives an algorithm from the two-way deterministic automata that runs, in the worst case, in $O(m + n)$ time for random sequences. The basic idea behind this algorithm is to avoid backtracking in the target string in the event of a mismatch, by taking advantage of information given by the type of mismatch. The target is processed sequentially from left to right. When a substring match-attempt fails, the previous symbols, which are known, are used to determine how far the pattern can be shifted to the right for the next match attempt.

Boyer and Moore[20] published a much faster algorithm in 1974. The speed of this algorithm is achieved by disregarding portions of the target that cannot, in any case, participate in a successful match.

The Naive Algorithm

The naive algorithm (Figure 2) is the simplest and most often used algorithm. It uses a linear and sequential character-based comparison at all positions in the text between y_0 and y_{n-m-1}, whether or not an occurrence of the pattern x starts at the current position. In case of success in matching the first element of the pattern x_0, each element of the pattern is successively tested against the text until failure or success

```
function naive(const target, pattern : PChar; const lTarget, lPattern : integer) : integer;
var
  i,                                          // main loop
  j    : integer;                             // comparison loop
begin
  result := -1;                               // returns -1 if no match or error
  if lTarget * lPattern = 0 then exit;        // nothing to compare
  for i := 0 to lTarget - 1 do                // main loop
    if pattern[0] = target[i] then
    begin
      for j := 1 to lPattern - 1 do                  // comparison loop
        if pattern[j] <> target[i+j] then break;     // failure, exit comparison loop
      if j = lPattern then                           // match found
      begin
        result := i+1;
        exit;
      end;
  end; // for
end; // Naive
```

Figure 2 The Naive algorithm.

```
function Karp_Rabin(const target, pattern : PChar; const lTarget, lPattern : integer) :
  integer;
const
  b = 8;                                      // func for bitwise left rotate
var
  hashPattern,                                // pattern hash value
  hashTarget,                                 // target hash value
  Bm,                                         // dummy var for computing the hash
  i,                                          // main loop
  j         : integer;
begin
  result := -1;
  if lTarget * lPattern = 0 then exit;
  Bm := 1;
  hashPattern := 0;
  hashTarget := 0;
                                              // preprocessing of pattern's hash value
  for j := 0 to lPattern-1 do
  begin
    Bm := Bm shl b;
    hashPattern := hashPattern shl b + ord(pattern[j]);
    hashTarget := hashTarget shl b + ord(target[j]);
  end;
                                                                  // search main loop
  for j := pred(lPattern) to pred(lTarget) do
  begin
    if (hashPattern = hashTarget) then                            // match of hash values
    begin
     i := 0;
     while (i < lPattern-1) and (pattern[i] = target[j-lPattern+1 + i]) do inc(i);
                                                        // comparison loop
     if i=lPattern - 1 then                            // Match found
     begin
         result := j-lPattern+2;
         exit;                                        // if failure, incremental computation
                                                      // of the target's hash value
     end   else   hashTarget   :=   hashTarget   shl   b   -   ord(target[j-lPattern+1])*Bm   +
  ord(target[j+1]);
    end   else   hashTarget   :=   hashTarget   shl   b   -   ord(target[j-lPattern+1])*Bm   +
  ord(target[j+1]);
  end; // for
end; // Karp_Rabin
```

Figure 3 The Karp-Rabin algorithm.

occurs at the last position. After each unsuccessful attempt, the pattern is shifted exactly one position to the right, and this procedure is repeated until the end of the target is reached.

The naive search algorithm has several advantages. It needs no preprocessing of any kind on the pattern and requires only a fixed amount of extra memory space.

The Karp-Rabin Algorithm

In theory, hashing functions provide a simple way to avoid the quadratic number of symbol comparisons in most practical situations. These functions run in constant time under reasonable probabilistic assumptions. Hashing technique was introduced by Harrison and later fully analyzed by Karp and Rabin. The Karp-Rabin algorithm (Figure 3) is a typical string-pattern-matching algorithm that uses the hashing technique.[21] Instead of checking at each position of the target to see whether the pattern occurs, it checks only whether the portion of the target aligned with the pattern has a hashing value similar to the pattern. To be helpful for the string-matching problem, great attention must be given to the hashing function. It must be efficiently computable, highly discriminating for strings, and computable in an incremental manner in order to decrease the cost of processing. Incremental hashing functions allow new hashing values for a window of the target to be computed step by step without the whole window having to be recomputed. The time performance of this algorithm is, in the unlikely worst case, $O(mn)$ with an expected time performance of $O(m + n)$.

Several different ways to perform the hashing function have been published. The original Karp-Rabin algorithm has been implemented in our system because it is easily adaptable to different alphabet sizes. The comparative measures we performed showed that most of the time is spent during the incremental hashing phase. Therefore, even though the Karp-Rabin algorithm needs fewer symbol comparisons than other algorithms to locate a pattern in a large target text, the cost of computing the hashing function outweighs the advantage of performing fewer symbol comparisons, at least for common medical language.

The Knuth-Morris-Pratt Algorithm

The first published linear-time string-matching algorithm was from Morris and Pratt and was improved by Knuth et al.[22] and others.[23] The Knuth-Morris-Pratt algorithm (Figure 4) is the classical algorithm that implements efficiently the left-to-right scan strategy. The search behaves like a recognition process by automaton, and a character of the target is compared to a character of the pattern. The basic idea behind the algorithm is to avoid backtracking in the target string in the event of a mismatch. This is achieved by the use of a failure function. When a substring match attempt fails, the previous character sequence is known (*the suffix*), and this fact can be exploited to determine how far to shift the pattern to the right for the next match attempt. Basically, if a partial match is found such that $target[i - j + 1..i - 1] = pattern[1..j - 1]$ and $target[i] \neq pattern[j]$, then matching is resumed by comparing $target[i]$ and $pattern[f(j - 1) + 1]$ if $j > 1$, where f is the failure function. If $j = 1$, then the comparison continues with $target[i + 1]$ and $pattern[1]$. An auxiliary step table (heuristic skip table) containing this optimal shift information (failure function) is computed from the pattern before the string is searched.

Two examples of failure function follow:

- Pattern "*abcabcacab*": If we are currently determining whether there is a match beginning at *target*[*i*], then if *target*[*i*] is not the character "*a*," we can proceed by comparing *target*[*i* + *1*] and "*a*." Similarly, if *target*[*i*] = "*a*" and *target*[*i* + *1*] < > "*b*," then we may continue by comparing *target*[*i* + *1*] and "*a*." If *target*[*i*] = "*a*" and *target*[*i* + *1*] = "*b*" = *ab* and *target*[*i* + *2*] < > "*c*," then we have the situation:

  target: "ab?.."

  where ? means that the character is not yet known.

- Pattern: "*abcabcacab*": The first ? in the target represents *target*[*i* + *2*] and it is different from the character "*c*." At this point, it is known that the search might be continued for a match by comparing the first character in the pattern with *target*[*i* + *2*]. However, there is no need to compare this character of the pattern with *target*[*i* + *1*], since we already know that *target*[*i* + *1*] is the same as the second character of the pattern, "*b*," and that it is not matched with the first character of the pattern, "*a*." Thus, by knowing the characters in the pattern and the position in the pattern where a mismatch occurs with a character in the target we can determine where in the pattern to continue the search for a match without moving backwards in the target.

The worst case performance occurs for normally distributed string patterns. In this case, the algorithm usually runs near $O(mn)$ in time complexity, but it may run in $O(m + n)$. This method performs well for large patterns made of repetitive short substrings.

```
function Knuth_Morris_Pratt(const target, pattern : PChar; const lTarget, lPattern :
  integer) : integer;
var
  step  : array[0..255] of integer;                    // failure table
  i,                                                   // main loop
  j     : integer;
begin
  result := -1;
  if lTarget * lPattern = 0 then exit;
  i := 0;
  j := -1;
  step[0] := -1;
                                                       // preprocessing the table
  repeat
    if (j = -1) or (pattern[i] = pattern[j]) then
    begin
      inc(i);
      inc(j);
      if pattern[j] = pattern[i] then step[i] := step[j] else step[i] := j;
    end else j := step[j];
  until i = lPattern - 1;
  j := -1;
  i := 0;
                                                       // search main loop
  while i < lTarget do
  begin
    if (j=-1) or (pattern[j] = target[i]) then         // comparison loop
    begin
      inc(i);
      inc(j);
      if j >= lPattern then                            // Match found
      begin
        result := i-j+1;
        exit;
      end;
    end else j := step[j];                             // skips the value found in the table
  end; // while
end; // Knuth_Morris_Pratt
```

Figure 4 The Knuth-Morris-Pratt algorithm.

This situation is rarely met in medical texts. Thus, in practice, the KMP algorithm is not likely to be significantly faster than the other approaches.

The Boyer-Moore-Horspool Algorithm

The Boyer-Moore algorithm (Figure 5) is considered the most efficient string-matching algorithm for natural language. In this method, the pattern is searched in the target from left to right. At each trial position, the symbol comparisons are performed to minimize the number of trials in case of unsuccessful matches while maximizing the pattern shift for the next trial. In a case of mismatch, the BM algorithm combines two different shift functions to optimize the number of characters that can be skipped during the skip process. These two shift functions are called the *bad-character shift* (or *occurrence* heuristic) and the *good-suffix shift* (or *match* heuristic). The latter method is similar to the one used in the KMP algorithm and is based on the pending symbol causing the mismatch. For every symbol of the pattern the length of a safe shift is stored. This shift corresponds to the distance between the last occurrence of that symbol in the pattern and the length of the pattern. The most interesting property of that technique is that the longer the length of the pattern, the longer the potential shifts. The final pattern shift is determined by taking the larger of the values from the two precomputed auxiliary tables. It is important to note that both methods can be preprocessed and kept in tables. Their time and space complexities depend only on the pattern. However, although there is a theoretic advantage in using the BM algorithm, many computational steps in this algorithm are costly in terms of processor instructions. The cost in time of the computational step was not shown to be amortized by the economy of character comparisons. This is particularly true of the function that computes the size comparisons in the skip tables. Horspool proposed a simplified form of the BM algorithm that use only a single auxiliary skip table indexed by the mismatching text symbols.[24] Baeza-Yates showed that the Boyer-Moore-Horspool algorithm (BMH) is the best in terms of average case perfor-

```
function Boyer_Moore_Horspool(const target, pattern : PChar; const lTarget, lPattern :
  integer) : integer;
var
  i,                                          // main loop
  j,                                          // comparison loop
  k       : integer;
  step    : array [0..255] of integer;        // skip table for the alphabet
begin
  result := 0;
  if lTarget * lPattern = 0 then exit;
  // preprocessing
  for k := 0 to 255 do step[k] := lPattern;          // initialize skip table
  for k := 0 to Pred(lPattern) do step[ord(pattern[k])] := lPattern-k;    // skip table for
                                                                         // the pattern
                                                     // search main loop
  k := Pred(lPattern);
  while k <= lTarget do
  begin
    i := k-1;
    j := lPattern-1;
    while target[i] = pattern[j] do                  // comparison loop
    begin
      dec(i);
      dec(j);
    end; // while
    if j = - 1 then                                  // Match found
    begin
      result := i + 2;
      exit;
    end;
    k := k + step[ord(target[k])];                   // if failure, skip the value in skip table
  end; // while
end; // Boyer_Moore_Horspool
```

Figure 5 The Boyer-Moore-Horspool algorithm.

mance for nearly all pattern lengths and alphabet sizes (remember that the French alphabet is somewhat larger than the English alphabe).[25] Yet, among all published algorithms we tested, the BMH algorithm showed the best performance in time, by far. The BMH algorithm will be studied more deeply, as it appears to be the best-choice algorithm in most cases. Because of its low space complexity, we recommend the use of the BMH algorithm in any case of exact string pattern matching, whatever the size of the target. Despite its apparent conceptual complexity, the BMH algorithm is relatively simple to implement. Moreover, it can easily be extended to handle multiple matches by generating events when a match occurs instead of leaving the algorithm as in the following example:

Example of pattern:

a	b	c	d

Example of text:

a	b	d	e	b	c	a	b	d	d	e	a	b	c	d

Construction of the skip table for the pattern:

a	b	c	d	e	f	…	y	z
3	2	1	4	4	4	4	4	4

This means that the distance from any character in the pattern to the last position in the pattern is known and that any character not in the pattern will produce a shift that is the length of the pattern. This table contains an entry for every single symbol present in the alphabet. For usual hardware platforms, it consists of the 256 entries found in an 8-bit ASCII table.

Search process

Loop 1: ← *right-to-left*

a	b	c	d	*pattern*[*i*]
			≠	
a	b	d	e	b	c	a	b	d	d	e	a	b	c	d	*target*[*j*]
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

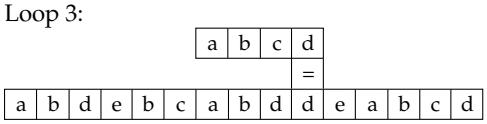
pattern[*i*] = *d* is different than *target*[*j*] = *e*.
Lookup in skip table for symbol *e* gives value 4; therefore, shift right four characters.

Loop 2: → *left-to-right shift*

				a	b	c	d	
							≠	
a	b	d	e	b	c	a	b	d	d	e	a	b	c	d
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

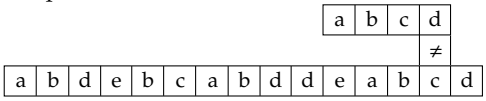
pattern[*i*] = *d* is different than *target*[*j*] = *b*.
Lookup in skip table for symbol *b* gives value 2; therefore, shift right two characters.

Loop 3:



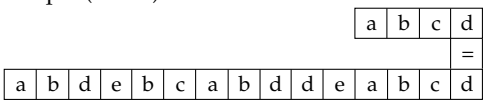
pattern[*i*] = *d* is equivalent to *target*[*j*] = *d*.
pattern[*i* – *1*] = *c* is different than *target*[*j* – *1*] = *d*.
Lookup in skip table for symbol *d* gives value 4; therefore, shift right four characters.

Loop 4:



pattern[*i* – *1*] = *d* is different than *target*[*j* – *1*] = *c*.
Lookup in skip table for symbol *c* gives value 1; therefore, shift right one character.

Loop 5 (finish):



Full match after four successful comparisons.

The BMH-2 algorithm is a slight variant of the BMH algorithm, which implements a new comparison method. In the BM and BMH algorithms, the comparison loop is entered if the last character of the pattern matches the current target position. In the BMH-2 algorithm, this loop is entered only if both the last and the middle characters of the pattern match with their respective positions in the target. This is a more conservative control of the right-to-left comparison loop. For long medical words, because of the frequent use of common suffixes and prefixes, this preliminary comparison generally allows skipping of several character comparisons and optimizes the left-to-right shift. Although this improvement decreases the number of character comparisons by 80 percent in the best cases, as described under Measures, the overall time complexity is often similar to that of the BMH algorithm. Given the increased time cost, performing the middle comparison becomes cost-effective only for long patterns, usually longer than 20 characters.

Space Complexity of the Tested Algorithms

For all algorithms, space usage of the pattern and the target is the same. The extra space complexity of each algorithm is self-explained in the code by the added variables—4 bytes for the naive algorithm, 12 bytes for the Karp-Rabin algorithm, 516 bytes for the Knuth-Morris-Pratt algorithm, and 518 bytes for the Boyer-Moore-Horspool algorithm. These values have been computed with 32-bit integers. They are constant.

Measures

Method

All algorithms were implemented in Object Pascal using Delphi version 4.03.[26] The tests were conducted on a 400-MHz Pentium II biprocessor PC with 256-Mbytes of 7ns SDRAM so that all tests could be done in memory. We used a biprocessor computer to minimize the variance of time measures caused by alternate system interruptions or operating system processes. All threads used by the test program were forced to run asynchronously on the same processor. Therefore, we expect the measures to be comparable with what could be found in real conditions in applications.

The target text was a corpus of medical texts that has a usual distribution of symbols and words. All tests were done using French and English texts. The French corpus consisted of 1,000 discharge summaries of the surgical depatment of the University Hospital of Geneva. The English corpus was the complete volume II of the World Health Organization's International Classification of Diseases, version 10, which was chosen because it is a stable, well-known, and reproducible source of text. This target text consisted of 197,550 words in 57,742 lines, corresponding to 1,606,861 characters (including spaces and end-of-line separators). All texts were converted to be compatible with the ASCII 8-bit table. The measures we present here were accomplished using the English corpus. Comparable results were found when using the French corpus.

All preprocessing loads were measured within their respective algorithms. The parameters of the function calls of each algorithm used pointers to zero-terminated strings in order to avoid variance due to memory management. All memory for strings was allocated before the calls. The functions, however, were self-contained. All function examples have been implemented using the same function call prototype —**function** *aSearchAlgorithm*(**const** target, pattern : PChar; **const** 1Target, 1Pattern : integer) : integer— where *pattern* and *target* are pointers to zero-terminated strings, *lTarget* and *lPattern* are integers representing the respective lengths of the target and the pattern. The functions return the value −1 in case of error, or else they return the position in the target where the first occurrence of pattern was found. No preprocessing or global variables were needed for their execution. When hashing tables were needed, the computation was done within the function and the time cost allocated to that function. Whenever possible, the functions were optimized for speed, for example, by using bitwise shifts to compute fast mul-

tiplications. Time measures were done using the motherboard's high-resolution performance counter. The expected precision is about 0.00083 msec. An overview of the method is shown in Figure 6.

The corpus of text was divided into 150 slices of 3,214 characters, except the last one, which had 39 fewer characters. All measurements were done in an incremental manner for a corpus size growing in 150 steps from the size of one slice to the whole target size. At each of the 150 increases in size, all algorithms were tested for the various patterns. Each measurement was repeated ten times at each increase step. The values reported are arithmetic means of these ten measurements.

If f is the position where the pattern can be found in the target, then a regular augmentation of duration will be observed as the size increases. This increase will be observed until the actual size reaches the f point. This increase of duration represents the processing cost and is a function of the size of both the pattern and the target. This function is equivalent to $O(f[n, m])$, the complexity in time. In all algorithms tested, this function is linear in time. A linear regression of this function permits comparison of the time complexity of the algorithms. The slope of the regression line corresponds to the average number of characters parsed per milliseconds.

Once the position f is reached, further increases of the target size will not continue to influence the time complexity. In the graphical representation of the results, the plot of the data shows a slope almost flat after the f position. The slight slopes remaining are due to the increase in space complexity of the target. This flat portion of the plot represents the stability of the algorithm to reach f even if the size after f continues to grow. The mean abscissa can be used to compare different algorithms with an independent t-test.

All the algorithms we considered work with an approximately constant cost in space. In the worst case, the extra space needed for processing is a linear function of the length of the pattern, which is negligible. Moreover, in this analysis, space complexity is similar for all algorithms tested. It must be emphasized that this would not be true for any algorithm. In particular, it excludes most automata based on tree representations that typically have a quadratic cost in space.

Choice of Patterns

All tests were performed using three different patterns. The first pattern tested was *dyspraxia*. This word was chosen because its position is at one third of the target. It is a short word (nine characters), composed of a very common prefix (*dys*) and a rather uncommon suffix (*praxia*). Such words tend to give an advantage to right-to-left algorithms.

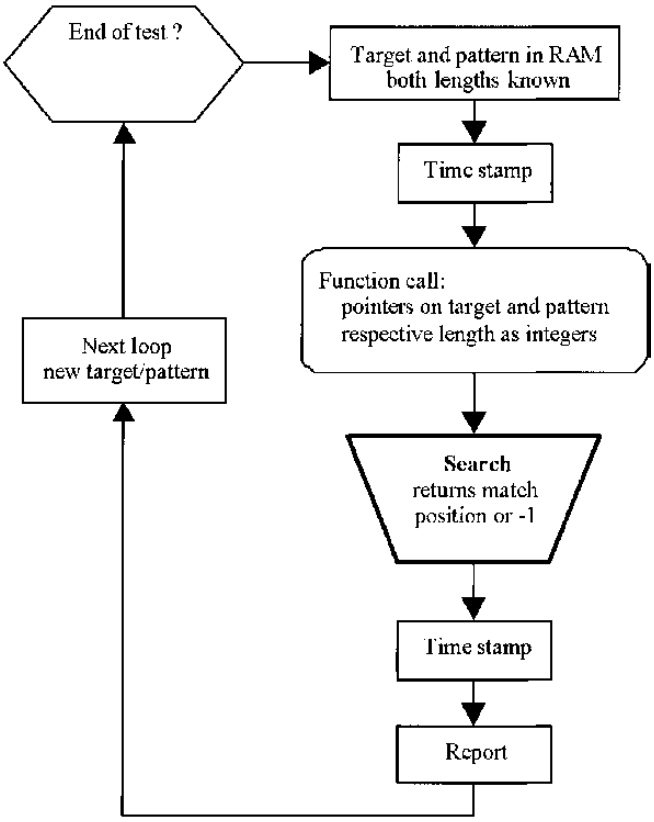


Figure 6 Overview of the test method.

The second pattern tested was *polychondritis*. It is a typical-length word (14 characters) for the medical domain, composed of three Latin/Greek roots (*poly–chondr–itis*). Among these roots, the first and the last are very common in the target corpus. For all types of algorithms, such words tend to increase the number of comparisons that fail. The length of this word tends to give a slight advantage to algorithms that use skip tables.

The last pattern tested was *superficial injuries of wrist and hand*. This sentence represents a typical aggregation of words used in the EPR. It is composed of six words and 37 characters. The length of such a pattern tends to favor algorithms that optimize the skip function. The length also favors algorithms that optimize the comparison loop. The pattern is located at the end of the target and therefore also demonstrates the impact of space complexity on overall performance.

Detailed data of the analysis and the program sources are available at http://www.lovis.net/pub/patterntest.htm

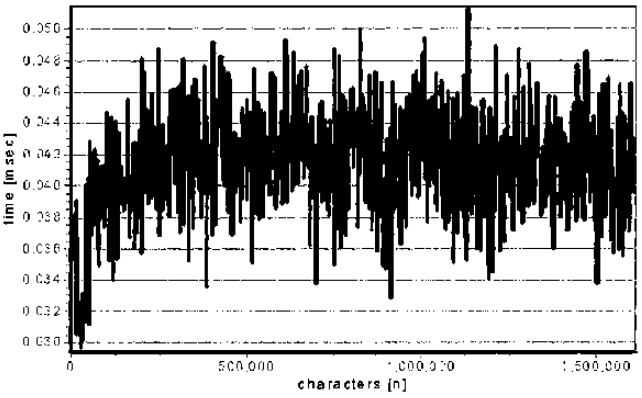


Figure 7 Baseline measures determining the noise of the system.

Statistical Analysis

All statistics were performed using the SPSS for Windows 8 statistical package. We used comparisons of groups with independent samples. Multiple linear regression was used, and all models were adjusted for noise introduced by the precision of the internal timer. For this purpose, 10^{12} measurements were performed using three dummy function calls with conditions and parameters similar to those that would have been used with the real functions. The plot of the data shows an initial increasing region that permitted a precise analysis of time complexity during mismatch phases. Analysis was performed using linear regression with the duration of processing as the predictor of interest and the size the target as the response variable. This made the interpretation of the slope easier. In this case, the slope was the number of characters analyzed in the target for each 1-msec increase of the predictor. Means were compared using an analysis of variance (ANOVA) and an independent two-tailed *t*-test. Where appropriate, 95% confidence intervals (CI) are given.

Results

Baseline

The baseline measurements of the system are shown in Figure 7. They can be assimilated to the noise of the system. The data are obtained by calling the function with parameters as used in normal conditions but without any action performed. Mean time spent to call function and background noise was 0.0414 ± 0.0032 msec, (95% CI, 0.04136–0.04166). These data are almost normally distributed. It must be emphasized that the cost in time is constant and independent of the size of the tested target, with the exception of a size fewer than 50 thousand characters, for which the cost in time is slightly less. This explains the slight left skewness of the histogram plot of these data. However, when these algorithms were evaluated in real conditions, this slight skewness appeared to be of negligible importance for the interpretation of the results.

Figure 8 shows an example of the graphical output of the data. The test was performed with the pattern *amnesia nos*. This pattern is found at position 647,906 in the English target (ICD-10). For all algorithms, the data presented the same global pattern. The first phase was a linear increase for duration along with augmentation of the size of the target, as long as the pattern was not present in the part of the target being analyzed. As soon as the pattern was found in the target, the duration remained approximately constant and independent of the size of the target that was after the match position. This figure shows that the time complexity increased linearly or nearly so for each algorithm until the target contained the pattern. On the one hand, analysis of slopes of the regression line before the match for algorithms using the same pattern allowed comparison of the efficiency of the algorithm for different target sizes. On the other hand, use of the same algorithm and different patterns allowed comparison of the efficiency according to the pattern size and provided, therefore, an experimental observation of the $O(f[n, m])$ theoretic evaluation.

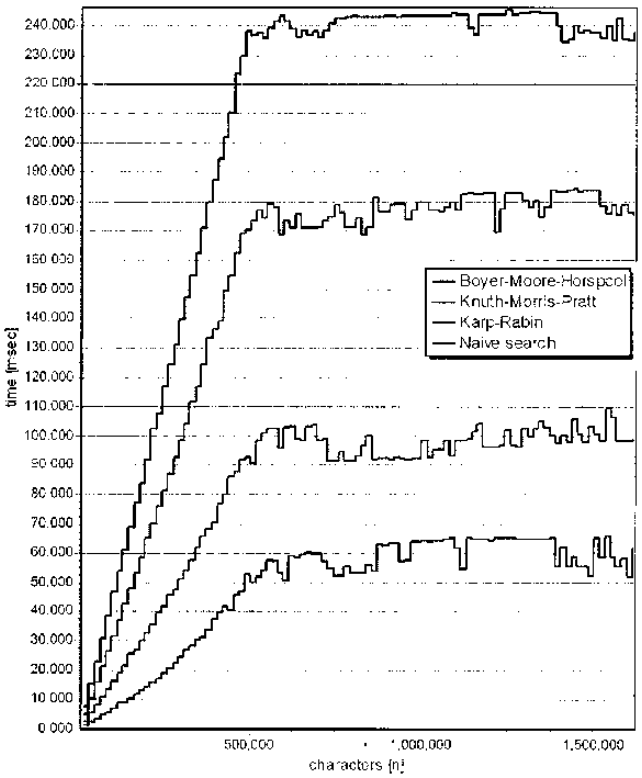


Figure 8 Complexity is linear in time.

Search for *Dyspraxia*

The first test was done using the word *dyspraxia*. This word was found at position 599,410 in the target text.

The difference in performance was highly significant for all group mean comparisons ($p < 0.0001$) except between BMH and BMH-2, where the p value was 0.661 (Table 2). It is worth noting that BMH is 1.68 times faster than the naive algorithm.

Search for *Polychondritis*

The word *polychondritis* was found at position 939,424 in the target text.

Means were very different between all groups ($p < 0.0001$) except BMH and BMH-2, where the p value was 0.99 (Table 3).

Search for *Superficial Injuries of Wrist and Hand*

The phrase *superficial injuries of wrist and hand* was found at position 1,224,835 in the target text.

Means were highly different between all groups ($p < 0.0001$), including BMH and BMH-2 (Table 4). However, the mean difference of 8.11 msec (95% CI, 6.18–10.03) in favor of BMH-2, while significant, was probably negligible in practice, since it represents less than 10 percent improvement.

Performance According to Pattern Size

Algorithms that do not use a skip table to optimize the shift function are rather independent of the pattern size in their time complexity. This was not the case for the two BMH algorithms. In the latter, one sees that the greater the pattern size, the longer the skip shift in case of mismatch and, therefore, the faster the algorithm. However, as illustrated with the word *polychondritis*, when the pattern ends with a suffix that appears frequently in the target, the BMH algorithms lose performance, because the loop comparison is more complex than the one in the naive algorithm. This fact was even more strongly illustrated with the BMH-2 algorithm, as a greater loss in the case of *polychondritis* and a greater win in the case of the longest pattern. Nevertheless, the BMH algorithms were still faster in all three cases (Table 5).

The overall results of all algorithms tested are presented in Figure 9, using the number of characters analyzed per millisecond. In this figure, for each pattern tested, the fastest algorithm received the index value of 1. The other algorithms for the same pattern are represented as fractions of the fastest. For example, for the pattern of nine characters, the KR algo-

Table 2 ■

Search Results Using the Word "Dyspraxia"

	Slope (char/msec)	Mean (msec)	SD (msec)	95% CI Mean (msec)
Naive	4,694.21	131.93	3.88	131.14–132.72
KMP	2,622.41	229.26	4.43	228.36–230.17
KR	2,008.03	296.63	4.40	295.73–297.53
BMH	8,648.89	77.71	5.83	76.52–78.90
BMH-2	8,051.44	77.30	6.90	75.90–78.71

NOTE: Naive indicates naive algorithm; KR, Karp-Rabin algorithm; KMP, Knuth-Morris-Pratt algorithm; BMH, Boyer-Moore-Horspool algorithm; BMH-2, variant of Boyer-Moore-Horspool algorithm.

Table 3 ■

Search Results Using the Word "Polychondritis"

	Slope (char/msec)	Mean (msec)	SD (msec)	95% CI Mean (msec)
Naive	4,491.43	199.25	4.45	198.13–200.38
KMP	2,514.84	365.95	4.27	364.88–367.03
KR	2,009.18	462.21	3.33	461.37–463.04
BMH	8,544.54	108.70	5.49	107.32–110.08
BMH-2	7,635.49	108.71	5.54	107.32–110.11

NOTE: Naive indicates naive algorithm; KR, Karp-Rabin algorithm; KMP, Knuth-Morris-Pratt algorithm; BMH, Boyer-Moore-Horspool algorithm; BMH-2, variant of Boyer-Moore-Horspool algorithm.

Table 4 ■

Search Results Using the Pattern "Superficial Injuries of Wrist and Hand"

	Slope (char/msec)	Mean (msec)	SD (msec)	95% CI Mean (msec)
Naive	4,295.91	273.58	3.28	272.47–274.49
KMP	2,570.51	468.25	3.12	467.19–469.30
KR	1,899.99	632.08	4.66	630.51–633.66
BMH	10,087.08	108.81	4.89	107.15–110.46
BMH-2	11,216.133	100.70	3.12	99.64–101.75

NOTE: Naive indicates naive algorithm; KR, Karp-Rabin algorithm; KMP, Knuth-Morris-Pratt algorithm; BMH, Boyer-Moore-Horspool algorithm; BMH-2, variant of Boyer-Moore-Horspool algorithm.

Table 5 ■

Slope Variation According to Pattern Size (Characters per Millisecond)

	9 chars/msec	14 chars/msec	37 chars/msec
Naive	4,694.21	4,491.43	4,295.91
KMP	2,622.41	2,514.84	2,570.51
KR	2,008.03	2,009.18	1,899.99
BMH	8,649.89	8,544.54	10,087.08
BMH-2	8,051.44	7,635.49	11,216.133

NOTE: Naive indicates naive algorithm; KR, Karp-Rabin algorithm; KMP, Knuth-Morris-Pratt algorithm; BMH, Boyer-Moore-Horspool algorithm; BMH-2, variant of Boyer-Moore-Horspool algorithm.

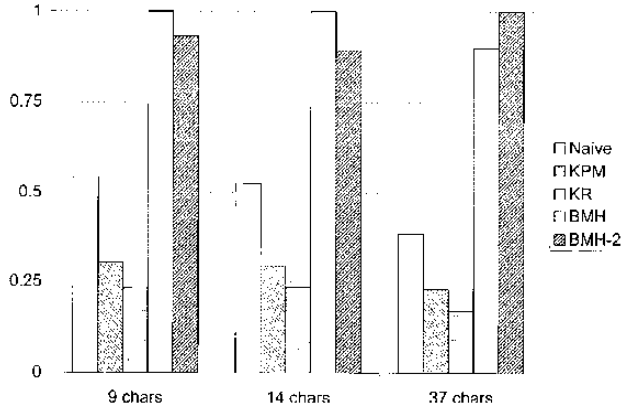


Figure 9 Overall performance of the five algorithms tested.

rithm performs the search at one fourth the speed of the BMH algorithm. This figure emphasizes the extreme speed of the BMH algorithm and the slight advantage to implement an optimized control for the optimization loop for long patterns. It shows also that the naive algorithm, compared with the BMH algorithm, performs less well with long patterns than with short ones.

Reproducibility

There are several limitations to this study. The performance of all algorithms tested may vary with the processors and the way they handle memory accesses, integer manipulation, and floating operations. On a given hardware, algorithms may behave differently according to the language used to implement them. In a given language, different versions of compilers may lead to different results. Finally, the programming style will affect the results. In our experience, however, it seems that the relative performance of each algorithm, one to the other, is stable across all these factors except the programmer's style.

The measures of performance have been performed on various texts in French and English. These texts include ICD-10, MEDLINE abstracts, discharge letters, and several nonmedical texts. The relative performance of all algorithms tested has been similar to that of the algorithms presented in this paper using ICD-10.

Conclusion

The BMH algorithm is a fast and easy-to-implement algorithm. The performance of this algorithm can barely be improved when working with medical lanaguage, except for long patterns. It typically performs better than the naive algorithm, which is mostly used for string pattern matching. Considering the growing amount of text handled in the EPR, the BMH algorithm is worth implementing in any case. Other algorithms that could theoretically perform better do not, compared with the BMH algorithm under real conditions. If long patterns are used, a more conservative control of the right-to-left comparison loop can slightly improve the time performance of the BMH algorithm.

References ■

1. Friedman C, Hripcsak G, Shagina L, Liu H. Representing information in patient reports using natural language processing and the extensible markup language. J Am Med Inform Assoc. 1999;6(1):76–87.
2. Stein HD, Nadkarni P, Erdos J, Miller PL. Exploring the degree of concordance of coded and textual data in answering clinical queries from a clinical data repository. J Am Med Inform Assoc. 2000;7:42–54.
3. Charlet J, Bachimont B, Brunie V, el Kassar S, Zweigenbaum P, Boisvieux JF. Hospitexte: toward a document-based hypertextual electronic medical record. Proc AMIA Symp. 1998:713–7.
4. Lovis C, Baud RH, Planche P. Power of expression in the electronic patient record: structured data or narrative text? Submitted to Int J Med Inform.
5. Lovis C, Baud RH, Scherrer JR. Internet integrated in the daily medical practice within an electronic patient record. Comput Biol Med. 1998;28:567–79.
6. Hume SC, Sunday D. Fast string searching. Software Practice and Experience. 1991;21(11):1221–48.
7. Sedgewick R. Algorithms. Reading, Mass: Addison-Wesley, 1983;19:241–55.
8. Forrest S. Genetic algorithms: principles of natural selection applied to computation. Science. 1993;261(5123):872–8.
9. Notredame C, Holm L, Higgins DG. COFFEE: an objective function for multiple sequence alignments. Bioinformatics. 1998;14(5):407–22.
10. Baud RH, Lovis C, Rassinoux AM, Scherrer JR. Morphosemantic parsing of medical expressions. Proc AMIA Symp. 1998;760–4.
11. Lovis C, Baud RH, Rassinoux AM, Michel PA. Medical dictionaries for patient encoding systems: a methodology. Artif Intel Med. 1998;14(1–2):201–14.
12. Pacak MG, Norton LM, Dunham GS. Morphosemantic analysis of *-itis* forms in medical language. Methods Inf Med. 1980(19):99–105.
13. Norton LM, Pacak MG. Morphosemantic analysis of compound word forms denoting surgical procedures. Methods Inf Med. 1983;(22):29–36.
14. Wolff S. The use of morphosemantic regularities in the medical vocabulary for automatic lexical coding. Methods Inf Med. 1984;(23):195–203.
15. Dujols P, Aubas P, Baylon C, Grémy F. Morphosemantic analysis and translation of medical compound terms. Methods Inf Med. 1991(30):30–5.
16. Hume SC, Sunday D. Fast string searching. Software Practice and Experience. 1991;21(11):1221–48.
17. Pirklbauer K. A study of pattern-matching algorithms. Structured Programming. 1992;(13):89–98.
18. Gonnet GH, Baeza-Yates R. Text Algorithms: Handbook of Algorithms and Data Structures in Pascal and C. 2nd edi-

tion. Wokingham, U.K.: Addison-Wesley, 1991(7):251–88.
19. Knuth DE, Morris JH Jr, Pratt VR. Fast pattern matching in strings. SIAM J Comput. 1997;6(1):323–50.
20. Boyer RS, Moore JS. A fast string searching algorithm. Commun ACM. 1977;20(10):762–72.
21. Karp RM, Rabin MO. Efficient randomized pattern-matching algorithms. IBM J Res Dev. 1987;31(2):249–60.
22. Knuth DE, Morris JH Jr, Pratt VR. Fast pattern matching in strings. SIAM J Comput. 1977;6(1):323–50.
23. Horowitz E, Sahni S. Fundamentals of Data Structures in Pascal. 4th ed. Woodland Hills, Calif: Computer Science Press, 1994:86–7.
24. Horspool RN. Practical fast searching in strings. Software Practice and Experience. 1980;10(6):501–6.
25. Baeza-Yates RA. Improved string matching. Software Practice and Experience. 1989;19(3):257–71.
26. Borland International. Delphi, version 4.03. Available at: http://www.borland.com or http://www.inprise.com.